Optimization-related extensions of the C++ language (single-threaded code)

Procedure integration

- Intel

```
#pragma inline/forceinline/noinline [recursive]
```

  - The forceinline pragma indicates that the calls in question should be inlined whenever the compiler is capable of doing so.
  - The inline pragma is a hint to the compiler that the user prefers that the calls in question be inlined, but expects the compiler not to inline them if its heuristics determine that the inlining would be overly aggressive and might slow down the compilation of the source code excessively, create too large of an executable, or degrade performance.
  - The noinline pragma indicates that the calls in question should not be inlined.

- gcc

```
void f() __attribute__((always_inline))
```

  - For functions declared inline, this attribute inlines the function independent of any restrictions that otherwise apply to inlining. Failure to inline such a function is diagnosed as an error.

```
void f() __attribute__((flatten))
```

  - Generally, inlining into a function is limited. For a function marked with this attribute, every call inside this function is inlined, if possible. Whether the function itself is considered for inlining depends on its size and the current inlining parameters.

```
void f() __attribute__((noinline))
```

Enforcing vectorization of vectorizable code

- Intel

```
#pragma vector always
#pragma simd
```

  - Asks the compiler to vectorize the loop if it is safe to do so, whether or not the compiler thinks that will improve performance.

- OpenMP

```
#pragma omp declare simd
```

- gcc

```
void f() __attribute__((simd))
```

  - This attribute enables creation of one or more function versions that can process multiple arguments using SIMD instructions from a single invocation.

Ignoring potential dependences

- pragma ivdep
  - Intel

```
#pragma ivdep
```

  - gcc

```
#pragma GCC ivdep
```

  - The ivdep pragma instructs the compiler to ignore assumed vector dependencies.
  - The proven dependencies that prevent vectorization are not ignored, only assumed dependencies are ignored.
  - In addition to the ivdep pragma, the vector pragma can be used to override the efficiency heuristics of the vectorizer.

```
void example(int *a, int k, int c, int m) {
  #pragma ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

Ignoring potential aliases

- SolarisCC (Oracle)

```
#pragma noalias (pointer, pointer [, pointer]…)

#pragma may_not_point_to (pointer, variable [, variable]…)
```

Ignoring potential aliases

- restrict - C99
  - The **restrict** keyword is part of the C standard since 1999
  - Not in C++ standard but most C++ compilers recognize **__restrict__**

```
void copy(int * restrict a, const int * b, int m) {
  for (int i = 0; i < m; i++)
    a[i] = b[i];
}
```

  - Semantics:
    - For any memory location modified directly or indirectly through a pointer marked by **restrict**, all read or write accesses to that location shall be made only through this pointer, directly or indirectly
    - This limitation is valid only for the duration of the block in which the pointer is declared
    - The formulation “directly or indirectly” includes variables initialized from this pointer
      - Any legal pointer value is always derived (by pointer arithmetic) from exactly one previously existing pointer value
    - It is the programmer’s responsibility to ensure that the limitation is not violated
    - The keyword has no effect if the pointer is only used in read operations
      - Only Read-Write, Write-Read and Write-Write dependences are relevant

Ignoring potential aliases

```
void copy(int * restrict a, const int * b, int m) {
  for (int i = 0; i < m; i++)        // this loop may be vectorized
  { int * ai = a + i;
    const int * bi = b + i;
    *ai = *bi;
  }
}
```

- Legal example

```
int m[100]; int * restrict x = m;    // this restrict actually does not improve anything
copy(x, x+10, 10); // correct wrt “restrict x” because both a and b are derived from x
copy(x+10, x+20, 10); // the compiler must not interleave the two invocations of copy
```

- Illegal examples

```
copy(x, x+10, 20); // violates “restrict a” because a[10] is read through b[0]
{ int * restrict y = x; // this restrict allows interleaving the two invocations of copy
  copy(y, x+10, 10);  // violates “restrict y” because y[10] is read through x[10]
  copy(y+10, x+20, 10);     // while y[10] is written here
}
```

Ignoring potential aliases

- The compiler will categorize all pointer variables and all memory accesses
  - One category for each “restrict” variable, one category for all the rest
  - Each memory access belongs to one or more categories

```
void copy(int * restrict a, const int * b, int m) {
  for (int i = 0; i < m; i++)
  { int * ai = a + i;                // ai belongs to [a]
    const int * bi = b + i;          // bi belongs to [others]
    *ai = *bi;                       // *ai does not alias with *bi in any iterations
  }
}
```

- Accesses in the same category are potentially aliased
  - Smart compilers may sometimes prove that such aliasing does not exist
    - E.g., if the address difference is a non-zero constant
  - In this example, understanding the loop control variable provides the proof
    - Therefore *ai does not alias with *ai in any other iteration
  - Together with the effect of restrict, this proof allows vectorization
    - No two writes can overlap, no write-read pair can overlap

Ignoring potential aliases

- Legal example – detailed discussion

```
int m[100]; int * restrict x = m;

copy(x, x+10, 10);

copy(x+10, x+20, 10);
```

- The copy procedures may be integrated into the caller, possibly after their vectorization

```
vectorized for (int i = 0; i < m; i++)

{ int * ai1 = x + i; const int * bi1 = x + 10 + i; *ai1 = *bi1; }

vectorized for (int i = 0; i < m; i++)

{ int * ai2 = x + 10 + i; const int * bi2 = x + 20 + i; *ai2 = *bi2; }
```

- All ai1,bi1,ai2,bi2 now belong to the same category of [x]
- Consequently, there are potential aliases between *ai1 and *bi2, as well as *bi1 and *ai2
- Thus, the compiler cannot interleave the loops:

```
// NOT AN EQUIVALENT CODE

vectorized for (int i = 0; i < m; i++)

{ int * ai1 = x + i; const int * bi1 = x + 10 + i; *ai1 = *bi1;

  int * ai2 = x + 10 + i; const int * bi2 = x + 20 + i; *ai2 = *bi2;

}
```

Ignoring potential aliases

- Illegal example – detailed discussion

```
int m[100]; int * x = m; int * restrict y = x;
copy(y, x+10, 10);  // violates “restrict y” because y[10] is read through x[10]
copy(y+10, x+20, 10);       // while y[10] is written here
```

- The copy procedures may be integrated into the caller, possibly after their vectorization

```
vectorized for (int i = 0; i < m; i++)
{ int * ai1 = y + i; const int * bi1 = x + 10 + i; *ai1 = *bi1; }
vectorized for (int i = 0; i < m; i++)
{ int * ai2 = y + 10 + i; const int * bi2 = x + 20 + i; *ai2 = *bi2; }
```

- ai1,ai2 belong to the category of [y], bi1,bi2 belong to [others]
- In addition, a smart compiler may prove that ai1 does not overlap with ai2
- As a result, the compiler may interleave the loops:

```
vectorized for (int i = 0; i < m; i++)
{ int * ai1 = y + i; const int * bi1 = x + 10 + i; *ai1 = *bi1;
  int * ai2 = y + 10 + i; const int * bi2 = x + 20 + i; *ai2 = *bi2;
}
```

- This is not an equivalent code – it is the programmer’s error to mark y with restrict

Ignoring potential aliases

- Advanced example – matrix transposition

```
void swap_ptr(int * a, int * b) { int t = *a; *a = *b; *b = t; }

void transpose_in_place(int * m, size_t n)

{ int * restrict um = m; int * restrict lm = m;

  for (int i = 0; i < n; ++i)

    for (int j = i + 1; j < n; ++j)

      swap(um + n*i + j, lm + n*j + i);

}
```

  - Then, swap will be integrated into transpose_in_place
- *a now belongs to the [um] category while *b to [lm]
  - Therefore, any *a access may be reordered wrt. *b accesses
- A smart compiler will determine that values of “a” in different iterations are different
  - The same property is true for “b” (provided n!=0)
  - Probably only within the inner loop; it is too difficult to prove for both loops
- Combined, anything may be reordered except that …
  - … reads of *a must precede the writes for the same a, similarly for b
- This allows vectorization of the *a reads and writes
  - Vectorization of *b reads and writes is allowed but difficult (gather/scatter required)